Exploring the Potential of Context-Aware Dynamic CPU Undervolting

Emmanouil Maroudas
emmmarou@uth.gr
Department of Electrical and Computer Engineering
Volos, Thessaly, Greece

Spyros Lalis
lalis@uth.gr
Department of Electrical and Computer Engineering
Volos, Thessaly, Greece

Nikolaos Bellas
nbellas@uth.gr
Department of Electrical and Computer Engineering
Volos, Thessaly, Greece

Christos D. Antonopoulos
cda@uth.gr
Department of Electrical and Computer Engineering
Volos, Thessaly, Greece

ABSTRACT

CPU operation at sub-nominal voltage levels has been researched to reduce the power and energy consumption of computer systems. While it is possible to determine a safe undervolting level for each application, typically only the most conservative setting is applied statically across all workloads. In this paper, we go a step further and investigate the gains that can be achieved by dynamically and transparently changing the level of CPU undervolting at runtime. To enable this functionality, we design and implement a novel, OS-level, context-aware dynamic undervolting mechanism, able to decide and apply voltage levels according to the specific tolerance of each workload that executes on a multicore CPU at a particular time. Our mechanism can further differentiate between the user- and kernel-level code executed within the same application thread, enabling the exploitation of differences in their undervolting potential. User- and kernel-level code have inherently different characteristics, yet in previous work have never been characterized individually. Our experiments, on an Intel x86-64 multicore show that the proposed approach can reduce the average CPU power consumption by 5.58%/30.05% compared to static undervolting and the nominal voltage level, respectively. Finally, we provide indicative estimates for the gains that could be achieved in future CPU architectures with multiple, per-core voltage domains.

CCS CONCEPTS

• **Software and its engineering** → **Power management**; • **Hardware** → *Platform power issues*; • **Computer systems organization** → Multicore architectures.

KEYWORDS

dynamic undervolting, energy awareness, power efficiency, workload characterization, system software

CF '21, May 11–13, 2021, Virtual Conference, Italy

ACM ISBN 978-1-4503-8404-9/21/05…$15.00
https://doi.org/10.1145/3457388.3458658

ACM Reference Format:
Emmanouil Maroudas, Spyros Lalis, Nikolaos Bellas, and Christos D. Antonopoulos. 2021. Exploring the Potential of Context-Aware Dynamic CPU Undervolting. In *Computing Frontiers Conference (CF '21), May 11–13, 2021,Virtual Conference, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3457388.3458658

1 INTRODUCTION

Energy efficiency is a first-class design goal across real-world systems of different scales, from large datacenters and HPC nodes to small embedded IoT devices and smartphones. This necessitates the design of more sophisticated, energy-aware systems [48] across the whole deployment range [3, 18, 19]. However, the shrinking of processor manufacturing designs towards smaller feature sizes aggravates the extent of energy and performance variability even across chips of the same family and production batch [31]. For this reason, the nominal frequency-voltage operating points of CPUs specified by the manufacturers include wide safety margins, in the order of 30% [11, 13]. They are, thus, inherently conservative and not optimized for energy efficiency [9].

There are many efforts to operate hardware at sub-nominal voltage levels without reducing frequency together with voltage (undervolting), as a method for exploiting the voltage margins [14] of CPUs [4, 5, 27, 35, 50], GPUs [28, 29, 44, 46, 49], RAMs [26, 47], FPGAs [41] and Wireless Sensor Nodes (WSNs) [25]. However, most of this work focuses on applying undervolting in a static way, based on the worst voltage constraint across all workloads of interest. This, in turn, limits the potential gains since it is not possible to exploit the fact that some applications can tolerate a higher degree of undervolting than others.

Moreover, existing work on CPU voltage margins characterization [35, 36] and exploitation [22, 23] treats the software stack monolithically, without distinguishing between different software components. In particular, no distinct characterization is performed for user-level code (application) and the kernel-level code of the underlying operating system, which runs on behalf of the application. This misses an additional opportunity to undervolt the CPU in a different manner during user- and system-level code execution.

In this paper, we go a step beyond the typical undervolting approaches and explore the potential of dynamic, context-aware CPU undervolting on real systems, without additional hardware support and with unmodified application binaries. To this end, we design

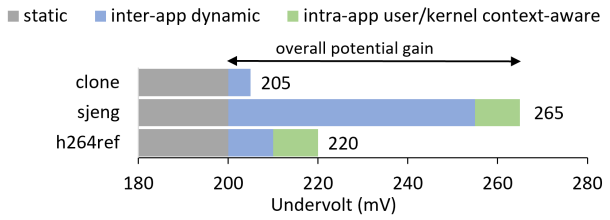


Figure 1: Undervolting tolerance (mV) of selected applications on an Intel processor. Horizontal axis starts at 180mV.

and implement CADU, an adaptive OS-level mechanism that dynamically adjusts voltage levels according to the specific constraints of the currently running workload. As an extension (CADU++), the mechanism can also distinguish between user- and kernel-level execution within the same thread. This makes it possible to characterize individually, and to exploit differences in undervolting tolerance between the user- and kernel-level code of applications.

This paper makes the following contributions: (i) We design and implement CADU/CADU++, an OS-level context-aware dynamic undervolting mechanism, which adapts CPU voltage at runtime, according to the undervolting tolerance of the currently running workload. (ii) We enable precise undervolting characterization, distinguishing between user- and kernel-level code, further extending the power-saving potential of CPU undervolting. (iii) We experimentally quantify the power-saving potential of CADU/CADU++ for both native and virtualized workloads. (iv) We identify the limitations of current processor designs in exploiting undervolting for energy efficiency, and we extrapolate the potential savings by context-aware CPU undervolting on future processor designs that may not have those limitations.

The rest of the paper is organized as follows. Section 2 discusses observations that motivate this work. Section 3 outlines our methodology and experimental setup. Section 4 introduces the design of CADU/CADU++. In Section 5 we experimentally quantify the undervolting tolerance of applications and the power-saving potential enabled by CADU/CADU++. In Section 6 we discuss the limitations of processors with a single voltage domain for all cores and provide power efficiency estimates for future platforms with multiple, per-core voltage domains (MVD). Section 7 outlines related work. Finally, Section 8 concludes the paper.

2 MOTIVATION

In static undervolting methods, the applications of interest are characterized separately to determine their individual tolerance to lower than nominal voltage. However, the system is ultimately set to operate at the sub-nominal voltage of the least tolerant application, even if this is not part of the currently running workload. While this approach still reduces the power / energy footprint of the system compared to operation at the nominal CPU voltage, it misses opportunities for additional savings due to workload variability.

A preliminary exploration of undervolting tolerance in Figure 1 reveals that different applications can operate at significantly different offsets from the nominal voltage level, even when running on the same processor. The deeper undervolting (blue part of the bars) that can be safely applied on top of the static setting (gray part of the bars) can be substantial. For example, the $sjeng$ benchmark can safely run at an undervolting level that is deeper by more than 25% w.r.t the static undervolting level of 200mV that would be required to support a wide range of applications. These extra margins can be exploited only via a dynamic undervolting approach.

In addition, as Figure 2 shows, kernel-time typically accounts for less than 1% of the total execution time for realistic workloads. It is therefore worth exploring whether one can increase the degree of undervolting for the user-level part of the execution, which accounts for more than 99% of the total execution time. Indeed, Figure 1 illustrates that the user-level (green part of the bars) can tolerate deeper undervolting, when characterized separately from the kernel part. Notably, for the $h264ref$ benchmark, this increases the opportunities of dynamic undervolting by more than $2x$.

The above observations motivate us to investigate dynamic undervolting on modern CPUs at application and privilege-level (user vs. kernel) granularity in a controlled fashion and to explore the power saving potential vs. the commonly studied static approach.

3 METHODOLOGY

We experiment on a system based on an Intel Skylake Xeon E3 v5 1220 processor, with DP of 80W, voltage range 550mV to 1520mV, and four cores on a single voltage domain (SVD). The operating system is Ubuntu 16.04 LTS running the 4.11 kernel, which we have extended to support dynamic undervolting in a context-aware manner (see Section 4). All experiments run at the maximum frequency of 3GHz with Turboboost disabled. Note that on modern Intel CPUs, system software can manipulate core voltage by using Model Specific Registers (MSRs) to specify a voltage offset with respect to the nominal voltage [32].

In our experiments, we utilize all four CPU cores. For single-threaded applications, each core runs an independent instance. For multi-threaded applications, we invoke a single instance across all cores. All multi-threaded workloads spawn worker threads (4 or more) to keep system utilization as close to 100% as possible (while not overloading the CPU). We characterize a subset of benchmarks from the stress-ng [20], PARSEC [8] and SPEC CPU2006[17] suites. These benchmarks are typically studied and characterized in many other works related to energy-awareness [16, 23, 34, 35, 38, 43, 48, 50]. They have diverse execution profiles, allowing us to evaluate the effects of operation at reduced CPU voltage on energy and power consumption over a wide and representative range of codes.

Initially, for each benchmark, we perform a separate, offline characterization to identify the workload-specific minimum safe voltage V_{min}, or equivalently the maximum voltage offset V_{off} that can be safely applied to the nominal CPU voltage. This is done through a binary search between the nominal V_{off} of 0mV and an overly aggressive, unsafe V_{off} of 300mV, at a resolution of 5mV. The characterization accepts the V_{off} tested as safe if the particular workload executes for 10 consecutive times without any errors. Up to this point, the characterization process is the same as for identifying the proper degree of undervolting in a static approach: this is the *narrowest* V_{off} across all benchmarks.

We then use the CADU++ mechanism to execute an additional characterization phase, in which we characterize separately the

Figure 2: Percentage of kernel-time over total execution time of benchmarks for native execution.

user- and kernel-space of each application. More specifically, we start from the V_{off} identified previously and progressively increase it (towards deeper undervolting) separately for the user- and kernel-level portions of the execution, to identify potential additional opportunities at either level. Parts of the entry/exit kernel code around CADU++ run at the user-level V_{off}, however, this has been evaluated as safe during characterization.

Overall, we find that the least tolerant measured V_{off} values are for *clone* (205mV) (Figure 1) and the *h264ref* (210mV), both in native (non-virtualized) execution. We round down the least tolerant level at 200mV and use it as the V_{off} for static undervolting.

We execute all our experiments on the same physical machine to provide a common reference for comparison, factoring out the effects of hardware variability. To apply our approach to different machines, these would need to be characterized separately, as is also the case in current static undervolting approaches. Indicative runs on machines with identical hardware show similar potential.

Given that virtualization is common practice in large-scale deployments, we replicate all experiments within virtual machines (VMs), on top of a QEMU/KVM-based hypervisor [6, 21], again on the same physical machine. In this case, our mechanism runs within the hypervisor, treating all code that runs within a VM as user-level and the hypervisor itself as system-level code. For a more fair comparison with the native execution results, we limit our evaluation only during workload execution, excluding any provisioning phases (VM boot, shutdown).

4 MECHANISM DESIGN & IMPLEMENTATION

In this section, we introduce CADU, our context-aware dynamic undervolting mechanism, built as an extension to the Linux kernel. This was developed to enable dynamic, transparent exploration of undervolting opportunities on top of a real hardware platform. We note that CADU controls and adjusts the voltage of the processor depending on the execution context, without changing frequency.

CADU conforms to the following design specifications: (i) It works on real-world hardware, without requiring specialized hardware support. (ii) It works with unmodified application binaries. (iii) The interaction with hardware is limited to a minimal, well-defined interface, thus making it easily portable to different architectures. (iv) The modifications required to the operating system (OS) are limited, making it portable to different OSs.

The main idea is to undervolt the processor to the most aggressive yet safe degree, subject to the tolerance of the thread that runs on each core at a given point in time. To this end, each thread is associated with an individualized pair of user-level / kernel-level V_{off}

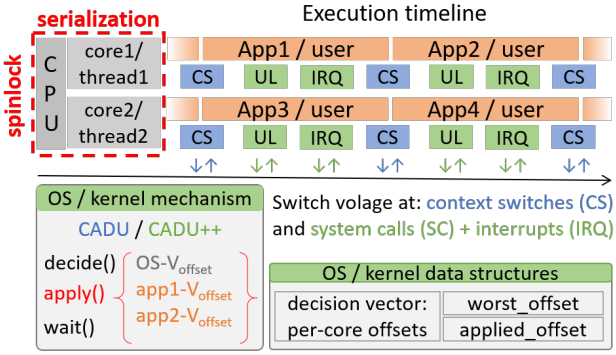


Figure 3: Design of CADU/ CADU++, with extended support for single voltage domain (SVD) processors.

values, identified via the characterization process discussed in Section 3. Obviously, executions with higher tolerance to undervolting are associated with larger V_{off}.

Figure 3 gives a high-level overview of the mechanism. Each core executes code independently, switching between application threads (context switch) and privilege levels (user-, kernel-level) through system call invocations or interrupts. Each of these entry / exit points triggers, through suitable hooks, the core in-kernel mechanism (down left). The mechanism (a) decides the next V_{off} based on the code that will execute next on the core, (b) applies it, and (c) waits to observe the new voltage being reached. The latter is done only if the applied V_{off} is smaller than the previous setting (thus configuration towards a higher voltage is commanded). This is to ensure that the voltage reaches the specified level before moving to a less tolerant execution context. Steps (b) and (c) are platform-specific. For Intel processors, such as the one used in our experiments, we perform voltage control through model specific registers (MSRs), as discussed in Section 3.

Notably, in modern x86-64 processors like the one used in our experiments, all cores are placed on a single (common) voltage domain (SVD), thus it is not possible for each core to operate at a different voltage. For this reason, a globally shared decision vector is used to record the current V_{off} preference of each core. This vector is consulted in step (a), choosing each time the least aggressive (smallest) V_{off} in order to keep the CPU within the undervolting tolerance of *all* active threads. This, in turn, mandates some additional synchronization between cores during steps (a) and (b) to avoid race

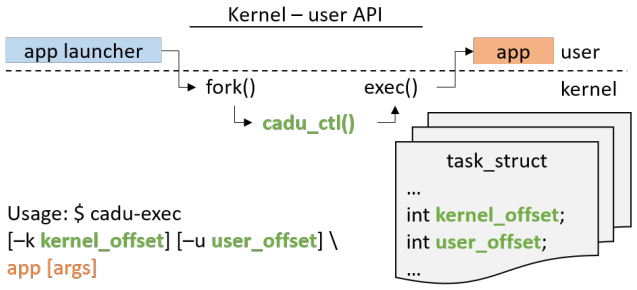


Figure 4: CADU application launcher

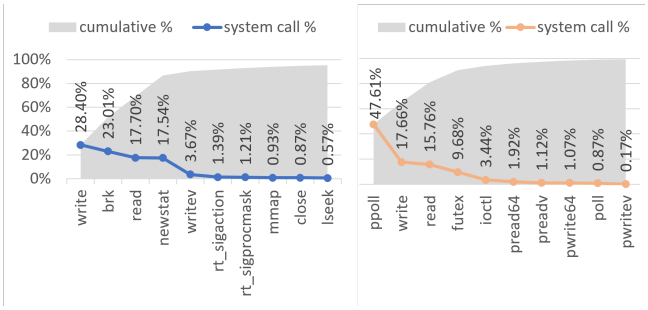


Figure 5: System call popularity (top 10): native (left) and virtualized (right) environment.

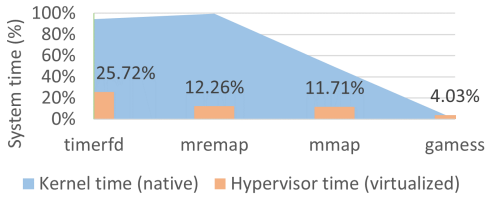


Figure 6: Percentage of hypervisor time over total execution (> 1%) of benchmarks (virtualized execution), compared to the respective kernel time percentage (native execution).

conditions that may lead to incorrect voltage adjustments. We implement a spinlock-based locking scheme that mutually excludes the cores from performing these steps concurrently. In Section 6, we discuss the SVD limitation, and explore the power / energy saving potential of CADU for future processor designs where cores may have separate voltage domains.

In order to inform the mechanism on the maximum tolerable offsets (user- and kernel-level) of each application, we use a simple application launcher (Figure 4) that essentially invokes a new system call (*cadu_ctl*) between *fork* and *exec*. Also, the mechanism can be configured to work only when switching between application threads (simple CADU) or also when switching between the user- and kernel-level code within the same thread (CADU++).

5 EXPERIMENTAL EVALUATION

In this Section, we analyze experimental results from workloads comprising multiple instances/threads of the same benchmark. In Section 6 we also discuss mixed, multi-benchmark workloads. In all cases, we execute as many instances/threads as required to maintain the CPU at full utilization. The widest tolerable V_{off} for each benchmark is identified via the characterization process discussed in Section 3. We use the *perf* tool [1] to quantify execution time and power consumption.

We focus on both native and virtualized executions. As Figures 5 and 6 illustrate, virtualizing a workload affects both the percentage of system-time, as well as the popularity of system calls. The former reflects on the effect of differences between the least tolerable voltage at the user- and system-level on average power consumption. The latter affects the tolerable voltage at the kernel-level since different code paths are excited.

5.1 Characterization of undervolting tolerance

Figure 7 quantifies the results for single application workloads at full utilization, for the native (top) and virtualized (bottom) execution. The stacked bars (corresponding to the left axis) show the degree of undervolting (V_{off} in mV) for static undervolting (static, gray color), when taking into account inter-application variability (CADU, blue), and when also taking into account intra-application user- / system-level variability (CADU++, green).

It is apparent that V_{off} is highly workload-dependent on both native and virtualized execution environments. An important observation is that during individual characterization of the user- and system-level of each application (CADU++), the undervolting potential of the user-level proved consistently higher than that of the kernel level (height of the green part of the bars). This is a strong indication that user-level code is typically more tolerant to undervolting than system-level hence the two levels should be characterized separately for optimal power gains.

Some indicative examples that illustrate the CADU++ improvement over unified user- / system-level undervolting within the same application (CADU) are *gamess*, *kill*, *mremap* and *dup* for the native execution and *str*, *timerfd*, *get* and *xalanbmk* for the virtualized execution. Examples where the execution environment affects the V_{off} of the workload can be observed for *get*, *timerfd* and *bodytrack*. In the native execution of these benchmarks, most of the V_{off} increase comes from the exploitation of inter-application variability (CADU), whereas the virtualized execution reaches to a similar depth of undervolting only when user- / system-level variability is taken into account (CADU++). Interestingly, there are also cases like *freqmine* and *memfd* where the unified user / system-level characterization is sufficient to reveal the full undervolting potential for both native and virtualized execution.

Figure 8 summarizes, across all benchmarks, the undervolting opportunities revealed by the characterization exploiting inter-application, as well as user- / system-level intra-application variability. On top of the static undervolting margin, exploiting inter-application variability (CADU) widens V_{off} by 37mV and 36mV on average for native and virtualized execution, respectively. Exploiting system- / user-level variability within each application (CADU++) further widens V_{off} by extra 8mV and 11mV on average for the native and virtualized execution.

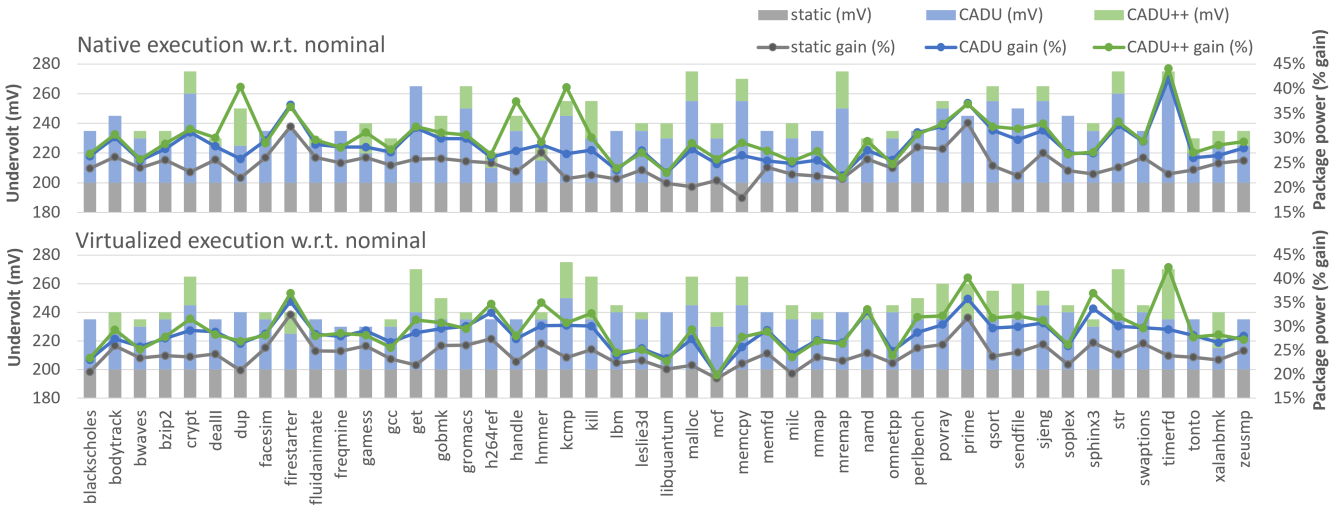


Figure 7: Voltage reduction (bars, left axis) and power gains (lines, right axis) compared with nominal execution. The left axis starts at 180mV. Workloads consist of multiple instances of the same benchmark (full CPU utilization) for native execution (top chart) and multiple VMs with the same benchmark (full CPU allocation) for the virtualized execution (bottom chart).

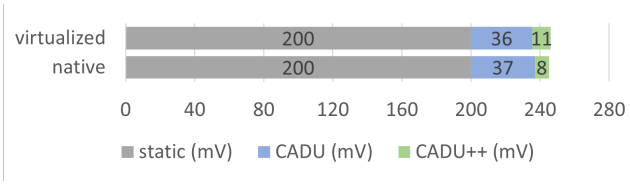


Figure 8: Average opportunities for undervolting (mV), across all benchmarks, w.r.t. nominal operating points.

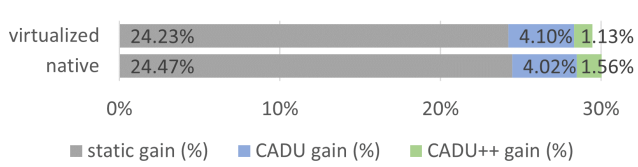


Figure 9: Average package power gain (%), across all benchmarks, w.r.t. power consumption when executing at nominal operating points.

5.2 Effect of dynamic undervolting on power efficiency

The lines in Figure 7 (corresponding to the right axis) illustrate the gain (%) of CPU package power w.r.t nominal operation, for the native (top) and virtualized (bottom) execution. We, again, distinguish between the gains of static undervolting, those achieved when taking into account inter-application variability (CADU), and by also taking into account intra-application user- / system-level variability (CADU++).

The highest power improvement of CADU over static is 19.14% in the native execution of the *timerfd* benchmark, and 6.76% in the virtualized execution of *kcmp*. The additional power improvement achieved by CADU++ in native executions is about 14% for *dup* and *kcmp*. In the virualized executions, the extra power gain is up to 13.02% for *timerfd*.

There are cases where although CADU++ improves the user-level V_{off}, this does not translate to power gains. An example of this situation is *mremap*. Although CADU++ improves the user-level V_{off} of the native execution by $25mV$ w.r.t CADU, this does not reflect to the overall power consumption. This is because this application spends 99.95% of its execution time inside the kernel, thus the extra user-level margins do not have any practical impact on the overall power consumption.

For the native execution, CADU++ shows marginal power improvement of about 0.37% versus CADU for *crypt*, *qsort*, *str*, *get* and *mremap*. Similarly, for the virtualized execution of *dup*, *handle*, *kcmp*, *memfd*, *mmap* and *mremap*, yielding a difference of merely ±0.15% on average. All these benchmarks have a significant percentage of kernel-time when executing natively; *mremap* and *mmap* also have high system-time ratio (~ 12%) in virtualized execution.

Figure 9 depicts the average power gain across all benchmarks when exploiting inter-application (CADU) and intra-application user- / system-level variability (CADU++). CADU reduces CPU power consumption by another ~ 4.10% on top of static, for both native and virtualized execution. CADU++ further improves CPU power consumption by 1.56% and 1.13% for the native and virtualized execution, respectively. The power gain with respect to static undervolting can reach up to 21.34% and 18.50% (*timerfd*) for the native and virtualized execution, respectively.

These results confirm our motivating observation that dynamic undervolting, in a controlled fashion, can indeed improve power consumption. We have also measured the power-to-the-plug and found this to be consistent with the gains reported by *perf*. In fact, the actual gains are larger due to the inefficiency of the power supply system. As we will discuss next, the overhead introduced

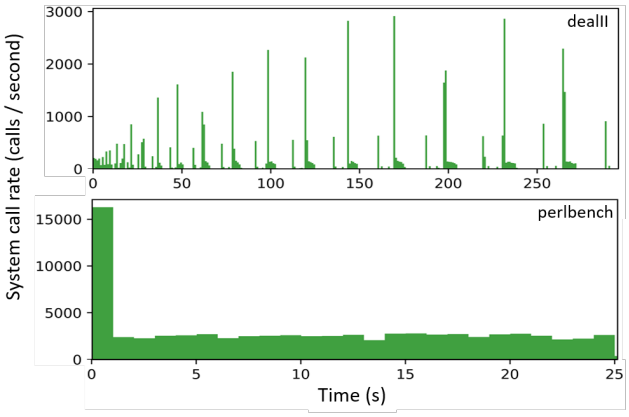


Figure 10: System call rate in the native execution of *dealII* and the virtualized execution of *perlbench*. Note that axis scales are different in the two plots.

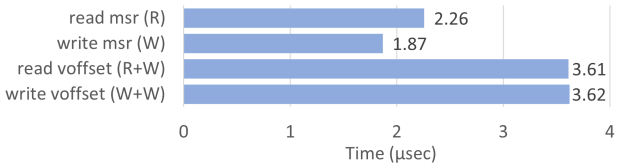
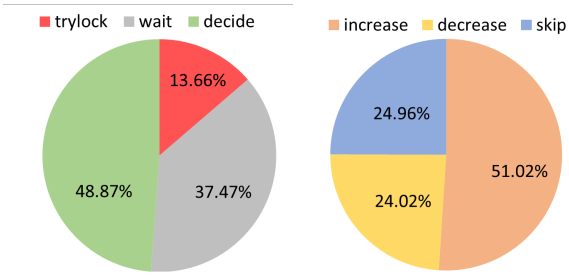


Figure 11: Average latency of voltage-control instruction sequences using Intel MSRs' interface.



(a) per-action **(b) per-decision**

Figure 12: Breakdown of CADU++ overhead (CPU cycles %).

by CADU++ is very small. Therefore, energy gains follow exactly the same trend and are of the same extent as power gains. Those results are omitted due to space limitations.

5.3 Overhead of dynamic voltage management by CADU++

The average performance overhead of CADU++ is quite minimal. Across all benchmarks, it is on average 0.45% and 0.79% for the native and virtualized execution, respectively.

There are two outliers with a significant performance overhead: the native execution of *dealII* (~ 8%) and the virtualized execution of *perlbench* (~ 10%). On the one hand, the system call rate of *dealII* (Figure 10 (top)) is characterized by multiple spikes of 2000 up to 3000 system calls per second throughout the entire execution, corresponding to invocations of *brk()* to allocate memory for the task. On the other hand, *perlbench* (Figure 10 (bottom)) has a steady high system call rate of 2200 to 2800 calls per second, mostly *stat* system calls. Also, during the first second of execution (initialization phase) there is a high spike of more than 15000 calls per second, mostly being *read*, *write* and *brk* system calls.

The high system call rate affects CADU++ overhead in two ways: (a) the mechanism is triggered very frequently (at all entry / exit points), and (b) due to the fact that the Intel Skylake architecture implements a single voltage domain for all cores, there is mutual exclusion – and thus serialization – at decision points where CADU++ needs to identify the least tolerant application out of those simultaneously executing on the cores of the CPU.

In virtualized executions, memory management is performed at a coarse granularity between the hypervisor and the guest OS. This explains why the high overhead of *dealII* is eliminated in virtualized execution. However, in virtualized executions, there is less memory per VM for filesystem caching, and the filesystem cache is not shared among different concurrently executing instances of the application (as this is the case in native execution). As a result, in the virtualized execution of *perlbench*, many filesystem calls eventually trap to the hypervisor.

We have measured the absolute cost of CPU voltage control operations through the MSR-based interface, using nanoBench [2]. The results are shown in Figure 11. Reading and writing MSRs has a latency of 2.26μsec and 1.87μsec respectively. Reading and writing the voltage offsets introduces a delay of approximately 3.60μsec.

We have also measured the relative overhead of the main operations within CADU++. Figure 12 illustrates this breakdown, as a percentage of the total CPU cycles spent in the execution of CADU++ code, for the worst-performing benchmark in native execution (*dealII*). Figure 12a breaks down the overhead to the main operations performed: waiting at serialization points (*trylock*), identifying the voltage offset to apply based on the least tolerant thread across all cores of the CPU (*decide*), and waiting for the voltage to reach the requested level when a voltage increase was commanded (*wait*). Note that *trylock* and *decide* account for more than 60% of the overhead, which could be eliminated in MVD architectures.

Figure 12b summarizes the overhead according to the "direction" of voltage manipulation. While the decision to maintain the same voltage (*skip*) is taken in 42.96% of CADU++ invocations, this contributes just 24.96% of the total overhead. In the rest of the invocations, the mechanism decides to *increase* and *decrease* the CPU voltage an equal number of times, each corresponding to 28.52% of the invocations. However, the relative performance overhead of *increase* is significantly higher, at 51.02% of total, due to waiting for the voltage to stabilize at the commanded level. In contrast, *decrease* accounts for just 24.02% of the overhead.

6 TOWARDS MVD ARCHITECTURES

The single cores voltage domain (SVD) of current modern processors may limit the gains of energy-saving mechanisms. In fact, Liu and Guo [30] show that multiple voltage domains can lead to increased energy gains. In our case, the SVD design hinders the full

exploitation of dynamic undervolting potential. This is because the V_{off} applied at any given time is the one of the *least* tolerant application executing on *any* of the cores of the processor. The same happens among user- and kernel-level V_{off}, particularly if the workload includes an application with high system call or interrupt rates, in which case the typically narrower kernel-level V_{off} (vs. the user-level V_{off}) will be applied to all cores.

If cores had separate voltage domains, a mechanism like CADU++ would be able to select the V_{off} independently for each core, further increasing the energy gain potential. In the following, we explore the gains that could be achieved on processor architectures with multiple voltage domains (MVD). Without loss of generality, we focus on native execution scenarios.

To quantify the extent of missed opportunities due to the single voltage domain (SVD), we experiment with the mixed workloads outlined in Table 1. They comprise applications with low undervolting tolerance and a narrow V_{off} (*h264ref*), applications with high tolerance and a wide V_{off} (*gromacs*, *sjeng*) and three randomly selected benchmarks (*dealII*, *namd* and *xalanbmk*). We let each workload run for 60 minutes. For every benchmark within the workload that finishes, a new instance is spawned.

Table 1: Mixed application workloads

Name	Benchmarks mix
mixed-run-1	1x h264ref, 3x gromacs
mixed-run-2	1x h264ref, 3x sjeng
mixed-run-3	1x (h264ref, dealII, namd, xalanbmk)

We use a simple analytical extrapolation model, to estimate the power consumption that would be expected under CADU++ on an architecture supporting multiple core voltage domains (MVD). The approach is explained in more detail below.

We first perform characterization experiments at full utilization, with single application workloads (let the target application be denoted by *app*), at different V_{off}^{app} levels, for each of the applications of Table 1. From each such experiment, we obtain package power $P_{pkg}^{V_{off}^{app}}$ and cores power $P_{cores}^{V_{off}^{app}}$ via the *perf* utility. Then, we calculate the power of the uncore component (all circuitry except the cores) of the processor as follows:

$$P_{uncore}^{V_{off}^{app}} = P_{pkg}^{V_{off}^{app}} - P_{cores}^{V_{off}^{app}} \tag{1}$$

The uncore power consumption obtained from Equation 1 across all V_{off}^{app} is between 10.40 and 11.08 Watts with an average value $P_{uncore}^{average}$ of 10.56 Watts. Therefore, we consider the uncore power to be workload-independent for our purposes and use the average value in our model. We should note that CADU++ does not alter the voltage of the uncore domain of the CPU.

We also make the assumption that, in the single application experiments, each core contributes equally to the cores' power as all cores are running the same benchmark. Thus, we estimate the power per core for each application and offset as follows:

$$P_{percore}^{V_{off}^{app}} = P_{cores}^{V_{off}^{app}} / ncores \tag{2}$$

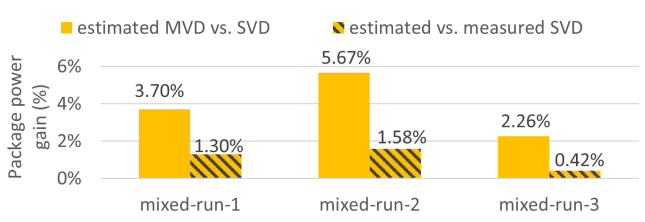


Figure 13: Estimated power consumption gain when moving from an SVD to an MVD architecture, for mixed application workloads. We also report the deviation of the estimated vs. measured power for our SVD processor.

where *ncores* is the number of cores. Based on the above, we estimate the overall package power consumption for potentially unseen combinations of applications and/or individual core voltages (assuming an MVD architecture), as follows:

$$P_{pkg}^{MVD} = P_{uncore}^{average} + \sum_{i=1}^{ncores} P_{percore}^{V_{off}^{app_i}} \tag{3}$$

where app_i is the application running on the i^{th} core and $V_{off}^{app_i}$ is the V_{off} of app_i, applied on that particular core.

For a single voltage domain (SVD) processor, where the mechanism is forced to pick the most conservative V_{off} across all applications that run on the cores of the processor, the package power consumption can be estimated as:

$$P_{pkg}^{SVD} = P_{uncore}^{average} + ncores \times P_{percore}^{\min_{1 \le i \le ncores} V_{off}^{app_i}} \tag{4}$$

To validate the accuracy of our model on our system, which has an SVD processor architecture, we experimentally quantify the CADU++ power consumption of the workloads of Table 1, for the voltage offsets of the least tolerant of those benchmarks, namely *h264ref* (user: 220mV, kernel: 210mV). Then, we estimate the power consumption of the same scenario using Equation 4, with $V_{off}^{app_i}$ equal to the *h264ref* setting, on all cores. We find that in all workload scenarios the model estimates the package power consumption of our SVD system within a deviation of less than 1.6% from the measured value, as shown in Figure 13.

As a next step, we use the model to estimate power consumption on a system with multiple core voltage domains (MVD). For each benchmark app_i in the mixed application workloads, we substitute the $P_{percore}^{V_{off}^{app_i}}$ term in Equation 3 with the respective power consumption of app_i identified in the characterization process for the highest V_{off}^{app} (lowest voltage) tolerated by that benchmark. Figure 13 shows that in this case CADU++ could further improve package power consumption up to 5.67% w.r.t. an SVD architecture. The MVD improvement mostly depends on the variation of the maximum tolerable V_{off} among the benchmarks participating in the mixed run, as well as between the user- and kernel-space tolerances of each benchmark and the percentage of time spent in the kernel. The potential for improvement would be even greater for CPUs with a larger number of cores. On an SVD architecture, it would be more probable for a single thread with low undervolting tolerance to constrain the whole CPU to a high voltage.

7 RELATED WORK

There are several studies that use machine learning to drive undervolting decisions. In [23], a methodology is introduced for predicting the undervolting tolerance of workloads at run-time, using information from performance counters. This information is then used as input to a dynamic voltage governor which adjusts CPU voltage at a sub-nominal level. The work in [15] simulates and studies a machine-learning approach at firmware level to predict the most energy-efficient supply voltage of various SoC components at runtime. Several other works evaluate the energy efficiency / resilience trade-offs. For instance, [45] performs processor undervolting to investigate this trade-off on HPC systems, while [16] and [12] explore the benefits of under-designed and opportunistic computing in processors and RAMs, respectively.

CADU++ is more context-aware than [15, 23]: it distinguishes between user- and kernel-level code and is aware of scheduling decisions. Moreover, compared with previous approaches, our approach is based on offline characterization. While this introduces an extra characterization step that needs to be performed before running the applications on the target system, it also achieves robustness to quick changes in the workload characteristics, which might necessitate prompt voltage adjustment (increase); in approaches that take dynamic undervolting decisions based on performance counters, such transitions may lead to crashes if not identified.

MREEF [10] is an online analysis framework that targets energy gains by reconfiguring multiple subsystems based on the identification of execution patterns. For CPUs, the reconfiguration is within the range of nominal frequency-voltage operating points. CADU is complementary to MREEF and can be combined with it as a more aggressive method towards energy efficiency. In [37], voltage margins are exploited from an approximate computing standpoint, trading off the quality of results for improved energy efficiency. Our approach reduces power consumption without having to resort to such trade-offs. It can also be used to exploit any additional voltage margins due to the approximation of complex computations.

There is also work on synergistic software/hardware approaches. [7] discusses so-called very low voltage (VLV) cross-layer designs to controlled undervolting from compilers to operating systems. [39] presents a compiler-based technique that controls the instruction rate issued by the application under aggressive operating margins, along with a collaborative hardware design that detects voltage emergencies and a fail-safe checkpoint mechanism. [42] proposes the use of path delay fault testing as a hardware-assisted method to select the best energy-efficient operating point, through software test routines that run periodically and provide critical path coverage. Unlike the above approaches, our work studies the potential power efficiency benefits on unmodified toolchains and binaries and without penalizing performance through frequency or instruction issue rate management. Moreover, our mechanism works on an off-the-shelf system and does not require special hardware support (beyond the means to control the degree of undervolting). It can also exploit MVD architectures, leading to additional gains.

Finally, there is a large body of work on early warning hardware mechanisms, which can improve the safety of undervolting even when attempting to go beyond the tolerance of the current workload. For ARMv8 server processors, ECC errors, SDC errors [35] and voltage droops [34] can provide insight into voltage prediction schemes and minimize failures. Critical path monitoring on POWER7 [27, 50] also improve the power efficiency. In [38] control flow and microarchitectural events are exploited to drive a voltage emergency predictor. AVOS [24] and TED-C [40] use hardware-assisted voltage overscaling until the number of detected hardware errors reaches a predefined threshold. ECC-guided voltage speculation on Itanium [4, 5] results to lower supply voltages and improved power consumption. Timing errors at circuit level can provide insight of the safety of the currently supplied voltage. Razor [13], is a dynamic voltage scaling design that can detect and eliminate static voltage margins [11]. BRAVO [43] proposes Balanced Reliability Metrics (BRM) as insight for processor reliability towards optimal supply voltage. In the processor family we study, there are no early signs of a possible system malfunction. When the voltage drops below the voltage that can be tolerated by the current workload, we observe a complete system crash, as in [23, 33]. For this reason, proper offline characterization is important to determine the degree of undervolting that can be safely applied to each application.

8 CONCLUSIONS

In this paper, we quantified the extent of the opportunities to improve the power-efficiency of CPUs by operating below their nominal voltage and dynamically changing the level of CPU undervolting at execution time, in a context-aware manner. Also, this is the first time where an attempt is made to characterize and exploit the difference in undervolting tolerance of the user- and kernel-level components of the same application.

To enable this study, we designed and implemented within the Linux kernel a proof of concept mechanism, which enables dynamic, transparent voltage management either when switching cores among applications (CADU), and when switching cores between the user- and kernel-level as well (CADU++). Our implementation introduces only a few and isolated modifications to the Linux kernel (only at entry and exit points), runs on off-the-shelf hardware (without any additional/special support), and operates on unmodified application binaries. Hardware-specific components are minimal and well-defined, enabling portability to different CPU architectures, including systems where each core (or group of cores) is placed in a separate voltage domain.

The experimental exploration using CADU/CADU++ on a modern SVD multi-core processor shows that considering both inter-application and user-/kernel-level variations, OSs can achieve more aggressive undervolting and increase power efficiency compared with existing monolithic approaches. We also show that the proposed approach is expected to achieve even greater gains in MVD processors supporting per-core voltage domains.

Overall, results suggest that integration of voltage control across the software stack is feasible and worth to be extended to all layers of the stack, paving the path towards more aggressive power management policies and mechanisms in future computing systems.

ACKNOWLEDGMENTS

This work has been partially funded by the European Commission, project UniServer, contract number 688540.

REFERENCES

[1] 2020. Perf tool. https://perf.wiki.kernel.org. [Online; accessed 26-July-2020].

[2] Andreas Abel and Jan Reineke. 2020. nanoBench: a low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 34–46.

[3] Massimo Alioto, Vivek De, and Andrea Marongiu. 2018. Energy-Quality Scalable Integrated Circuits and Systems: Continuing Energy Scaling in the Twilight of Moore's Law. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 8, 4 (Dec. 2018), 653–678. https://doi.org/10.1109/jetcas.2018.2881461

[4] Anys Bacha and Radu Teodorescu. 2013. Dynamic reduction of voltage margins by leveraging on-chip ECC in Itanium II processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13*. ACM Press. https://doi.org/10.1145/2485922.2485948

[5] Anys Bacha and Radu Teodorescu. 2014. Using ECC Feedback to Guide Voltage Speculation in Low-Voltage Processors. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. https://doi.org/10.1109/micro.2014.54

[6] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[7] Ramon Bertran, Pradip Bose, David Brooks, Jeff Burns, Alper Buyuktosunoglu, Nandhini Chandramoorthy, Eric Cheng, Martin Cochet, Schuyler Eldridge, Daniel Friedman, et al. 2017. Very low voltage (VLV) design. In *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 601–604.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

[9] S. Chandra, K. Lahiri, A. Raghunathan, and S. Dey. 2009. Variation-Tolerant Dynamic Power Management at the System-Level. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 9 (Sept. 2009), 1220–1232. https://doi.org/10.1109/tvlsi.2009.2019803

[10] Ghislain Landry Tsafack Chetsa, Laurent Lefevre, Jean-Marc Pierson, Patricia Stolf, and Georges Da Costa. 2015. Application-Agnostic Framework for Improving the Energy Efficiency of Multiple HPC Subsystems. In *2015 23rd Euromicro International Conference on Parallel Distributed, and Network-Based Processing*. IEEE. https://doi.org/10.1109/pdp.2015.18

[11] Shidhartha Das, David Roberts, Seokwoo Lee, Sanjay Pant, David Blaauw, Todd Austin, Krisztián Flautner, and Trevor Mudge. 2006. A self-tuning DVS processor using delay-error detection and correction. *IEEE Journal of Solid-State Circuits* 41, 4 (2006), 792–804.

[12] N. Dutt, P. Gupta, A. Nicolau, L. A. D. Bathen, and M. Gottscho. 2013. Variability-aware memory management for nanoscale computing. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. https://doi.org/10.1109/aspdac.2013.6509584

[13] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, Toan Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. [n.d.]. Razor: a low-power pipeline based on circuit-level timing speculation. In *22nd Digital Avionics Systems Conference. Proceedings (Cat. No.03CH37449)*. IEEE Comput. Soc. https://doi.org/10.1109/micro.2003.1253179

[14] Dimitris Gizopoulos, George Papadimitriou, Athanasios Chatzidimitriou, Vijay Janapa Reddi, Behzad Salami, Osman S Unsal, Adrian Cristal Kestelman, and Jingwen Leng. 2019. Modern Hardware Margins: CPUs, GPUs, FPGAs Recent System-Level Studies. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 129–134.

[15] Mohammad Saber Golanbari and Mehdi B. Tahoori. 2018. Runtime Adjustment of IoT System-on-Chips for Minimum Energy Operation. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE. https://doi.org/10.1109/dac.2018.8465782

[16] Puneet Gupta, Yuvraj Agarwal, Lara Dolecek, Nikil Dutt, Rajesh K. Gupta, Rakesh Kumar, Subhasish Mitra, Alexandru Nicolau, Tajana Simunic Rosing, Mani B. Srivastava, Steven Swanson, and Dennis Sylvester. 2013. Underdesigned and Opportunistic Computing in Presence of Hardware Variability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 1 (Jan. 2013), 8–23. https://doi.org/10.1109/tcad.2012.2223467

[17] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[18] Christos Kalogirou, Panos Koutsovasilis, Christos D. Antonopoulos, Nikolaos Bellas, Spyros Lalis, Srikumar Venugopal, and Christian Pinto. 2019. Exploiting CPU Voltage Margins to Increase the Profit of Cloud Infrastructure Providers. In *2019 19th IEEE/ACM International Symposium on Cluster Cloud and Grid Computing (CCGRID)*. IEEE. https://doi.org/10.1109/ccgrid.2019.00044

[19] Georgios Karakonstantis, Konstantinos Tovletoglou, Lev Mukhanov, Hans Vandierendonck, Dimitrios S. Nikolopoulos, Peter Lawthers, Panos Koutsovasilis, Manolis Maroudas, Christos D. Antonopoulos, Christos Kalogirou, Nikos Bellas, Spyros Lalis, Srikumar Venugopal, Arnau Prat-Perez, Alejandro Lampropulos, Marios Kleanthous, Andreas Diavastos, Zacharias Hadjilambrou, Panagiota Nikolaou, Yiannakis Sazeides, Pedro Trancoso, George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Dimitris Gizopoulos, and Shidhartha Das. 2018. An energy-efficient and error-resilient server ecosystem exceeding conservative scaling limits. In *2018 Design Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. https://doi.org/10.23919/date.2018.8342175

[20] Colin Ian King. 2017. Stress-ng. *URL: http://kernel. ubuntu. com/git/cking/stressng. git/(visited on 28/03/2018)* (2017).

[21] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*.

[22] P. Koutsovasilis, C. Antonopoulos, N. Bellas, S. Lalis, G. Papadimitriou, A. Chatzidimitriou, and D. Gizopoulos. 2020. The Impact of CPU Voltage Margins on Power-Constrained Execution. *IEEE Transactions on Sustainable Computing* (2020), 1–1. https://doi.org/10.1109/TSUSC.2020.3045195

[23] Panos Koutsovasilis, Konstantinos Parasyris, Christos D Antonopoulos, Nikolaos Bellas, and Spyros Lalis. 2020. Dynamic Undervolting to Improve Energy Efficiency on Multicore X86 CPUs. *IEEE Transactions on Parallel and Distributed Systems* 31, 12 (2020), 2851–2864.

[24] P K Krause and I Polian. 2011. Adaptive voltage over-scaling for resilient applications. In *2011 Design Automation & Test in Europe*. IEEE. https://doi.org/10.1109/date.2011.5763153

[25] Ulf Kulau, Felix Büsching, and Lars Wolf. 2016. IdealVolting. *ACM Transactions on Sensor Networks* 12, 2 (April 2016), 1–38. https://doi.org/10.1145/2885500

[26] Seyed Saber Nabavi Larimi, Behzad Salami, Osman S. Unsal, Adrian Cristal Kestelman, Hamid Sarbazi-Azad, and Onur Mutlu. 2020. Understanding Power Consumption and Reliability of High-Bandwidth Memory with Voltage Underscaling. arXiv:2101.00969 [cs.AR]

[27] Charles R. Lefurgy, Alan J. Drake, Michael S. Floyd, Malcolm S. Allen-Ware, Bishop Brock, Jose A. Tierno, and John B. Carter. 2011. Active management of timing guardband to save energy in POWER7. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11*. ACM Press. https://doi.org/10.1145/2155620.2155622

[28] Jingwen Leng, Alper Buyuktosunoglu, Ramon Bertran, Pradip Bose, and Vijay Janapa Reddi. 2015. Safe limits on voltage reduction efficiency in GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*. ACM Press. https://doi.org/10.1145/2830772.2830811

[29] Jingwen Leng, Yazhou Zu, and Vijay Janapa Reddi. 2014. Energy Efficiency Benefits of Reducing the Voltage Guardband on the Kepler GPU Architecture. (2014).

[30] Jun Liu and Jinhua Guo. 2016. Energy efficient scheduling of real-time tasks on multi-core processors with voltage islands. *Future Generation Computer Systems* 56 (March 2016), 202–210. https://doi.org/10.1016/j.future.2015.06.003

[31] Aniruddha Marathe, Yijia Zhang, Grayson Blanks, Nirmal Kumbhare, Ghaleb Abdulla, and Barry Rountree. 2017. An empirical survey of performance and energy efficiency variation on Intel processors. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing - E2SC'17*. ACM Press. https://doi.org/10.1145/3149412.3149421

[32] Mihic. [n.d.]. mihic/linux-intel-undervolt. https://github.com/mihic/linux-intel-undervolt

[33] N. Pandit, Z. Kalbarczyk, and R. K. Iyer. 2009. Effectiveness of machine checks for error diagnostics. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. 578–583. https://doi.org/10.1109/DSN.2009.5270290

[34] George Papadimitriou, Athanasios Chatzidimitriou, and Dimitris Gizopoulos. 2019. Adaptive Voltage/Frequency Scaling and Core Allocation for Balanced Energy and Performance on Multicore CPUs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. https://doi.org/10.1109/hpca.2019.00033

[35] George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Dimitris Gizopoulos, Peter Lawthers, and Shidhartha Das. 2017. Harnessing voltage margins for energy efficiency in multicore CPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-50 '17*. ACM Press. https://doi.org/10.1145/3123939.3124537

[36] George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Charalampos Magdalinos, and Dimitris Gizopoulos. 2017. Voltage margins identification on commercial x86-64 multicore microprocessors. In *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE. https://doi.org/10.1109/iolts.2017.8046198

[37] Konstantinos Parasyris, Vassilis Vassiliadis, Christos D. Antonopoulos, Spyros Lalis, and Nikolaos Bellas. 2017. Significance-Aware Program Execution on Unreliable Hardware. *ACM Transactions on Architecture and Code Optimization* 14, 2 (April 2017), 1–25. https://doi.org/10.1145/3058980

[38] Vijay Janapa Reddi, Meeta S. Gupta, Glenn Holloway, Gu-Yeon Wei, Michael D. Smith, and David Brooks. 2009. Voltage emergency prediction: Using signatures to reduce operating margins. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE. https://doi.org/10.1109/hpca.2009.4798233

[39] Vijay Janapa Reddi, Meeta S. Gupta, Michael D. Smith, Gu yeon Wei, David Brooks, and Simone Campanoni. 2009. Software-assisted hardware reliability. In *Proceedings of the 46th Annual Design Automation Conference on ZZZ - DAC '09*. ACM Press. https://doi.org/10.1145/1629911.1630114

[40] Roberto Giorgio Rizzo and Andrea Calimera. 2017. Tunable Error Detection-Correction for Efficient Adaptive Voltage Over-Scaling. In *2017 New Generation of CAS (NGCAS)*. IEEE. https://doi.org/10.1109/ngcas.2017.75

[41] Behzad Salami, Osman S. Unsal, and Adrian Cristal Kestelman. 2018. Comprehensive Evaluation of Supply Voltage Underscaling in FPGA on-Chip Memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. https://doi.org/10.1109/micro.2018.00064

[42] John Sartori and Rakesh Kumar. 2014. Software canaries. In *Proceedings of the 2014 international symposium on Low power electronics and design - ISLPED '14*. ACM Press. https://doi.org/10.1145/2627369.2627646

[43] Karthik Swaminathan, Nandhini Chandramoorthy, Chen-Yong Cher, Ramon Bertran, Alper Buyuktosunoglu, and Pradip Bose. 2017. BRAVO: Balanced Reliability-Aware Voltage Optimization. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. https://doi.org/10.1109/hpca.2017.56

[44] Jingweijia Tan, Shuaiwen Leon Song, Kaige Yan, Xin Fu, Andres Marquez, and Darren Kerbyson. 2016. Combating the Reliability Challenge of GPU Register File at Low Supply Voltage. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation - PACT '16*. ACM Press. https://doi.org/10.1145/2967938.2967951

[45] Li Tan, Shuaiwen Leon Song, Panruo Wu, Zizhong Chen, Rong Ge, and Darren J. Kerbyson. 2015. Investigating the Interplay between Energy Efficiency and Resilience in High Performance Computing. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. https://doi.org/10.1109/ipdps.2015.108

[46] Renji Thomas, Kristin Barber, Naser Sedaghati, Li Zhou, and Radu Teodorescu. 2016. Core tunneling: Variation-aware voltage noise mitigation in GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. https://doi.org/10.1109/hpca.2016.7446061

[47] Konstantinos Tovletoglou, Lev Mukhanov, Georgios Karakonstantis, Athanasios Chatzidimitriou, George Papadimitriou, Manolis Kaliorakis, Dimitris Gizopoulos, Zacharias Hadjilambrou, Yiannakis Sazeides, Alejandro Lampropulos, Shidhartha Das, and Phong Vo. 2018. Measuring and Exploiting Guardbands of Server-Grade ARMv8 CPU Cores and DRAMs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE. https://doi.org/10.1109/dsn-w.2018.00013

[48] Lucas Wanner, Liangzhen Lai, Abbas Rahimi, Mark Gottscho, Pietro Mercati, Chu-Hsiang Huang, Frederic Sala, Yuvraj Agarwal, Lara Dolecek, Nikil Dutt, Puneet Gupta, Rajesh Gupta, Ranjit Jhala, Rakesh Kumar, Sorin Lerner, Subhasish Mitra, Alexandru Nicolau, Tajana Simunic Rosing, Mani B. Srivastava, Steve Swanson, Dennis Sylvester, and Yuanyuan Zhou. 2015. NSF expedition on variability-aware software: Recent results and contributions. *it - Information Technology* 57, 3 (Jan. 2015). https://doi.org/10.1515/itit-2014-1085

[49] Hadi Zamani, Yuanlai Liu, Devashree Tripathy, Laxmi Bhuyan, and Zizhong Chen. 2019. GreenMM: Energy Efficient GPU Matrix Multiplication through Undervolting. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. ACM Press, New York, NY, USA, 308–318. https://doi.org/10.1145/3330345.3330373

[50] Yazhou Zu, Charles R. Lefurgy, Jingwen Leng, Matthew Halpern, Michael S. Floyd, and Vijay Janapa Reddi. 2015. Adaptive guardband scheduling to improve system-level efficiency of the POWER7+. In *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*. ACM Press. https://doi.org/10.1145/2830772.2830824